Center for Applied Autonomous Sensor Systems (AASS)
Örebro University, School of Science and Technology
SE-70182 Örebro
Sweden

OOP Lab I Extra: Advanced Git

It turns out, one can have a lot of fun comiting untested code into the master branch. In this lab we will take a look at advanced git branching and how to avoid common problems.

This lab keeps numbering continuously with the previous lab instructions.

Please note, this extra part of the Lab 1 is largerly based on self education. You are expected to search for information when instructions are not clear.

The primary place to search for info and tutorials are as follows:

- https://www.atlassian.com/git/tutorials
- https://git-scm.com/doc

Part 4: Theory of Git

Basics

Git was developed by Linux Torvalds and his team around 2005 to handle version control of one very important thing: Linux kernel.

It is estimated that as of 2019, more than 70% of all repositories use git as a version control systems. The fact is that you absolutely have to be familiar with git if looking for a serious software development job.

It might not be very practical for you now to lean how exactly git works. Instead, we will look into what it does.

So, the most important thing is that git allows you to keep different versions of your code. When you add some new feature or bug to your code, you make a *commit* and git remembers what exacty has been changed. So, commit-by-commit you develop your software and you can always reset or revert what you have done. You can also browse the history of your software to see and reflect on how it was progressing.

So, commit is a smallest possible unit of work in git, like some feature in your sofwtare. However, it has to be meaningful, avoid making commits for every single line of code. Sometimes, you have to make many commits to try out some new solution and introduce a very sophisticated bug. In this case, you can create a separate *branch* from you existing code and work on it until you finish implementation. Then, the code from branch can be *merged* into your main code.

Usially (but not necessarily), you main branch in the repository is called “master”. You can create any branch names as long as you follow branch naming conventions.

Please note, git works with folders and text files. It does not know anything about programming languages you are using or you coding paradigms. It's just looking for changes in your files and folders. So, any git repository is essentially a folder (or directory) in your computer. You can make any directory a repository by using *git init* command in it.

Git is not really good at tracking changes in big binary files (like videos), like videos. But it can do it if it's needed, so keep an eye on what files you are actually storing in git repository.

To wrap it up, if you want to have a local-only repository (folder with source code) on your computer, you first need to initialize it with *git init,* then add files to be tracked by *git add* and then commit changes using *git commit* command. For working with branches, you need to look for *git branch*, *git checkout*, and *git merge* commands. Please, search the docs for exact syntax of commands.

Collaboration

You can use git as a version control for some folder on your computer. But git can also be use as a very powerfull collaboration tool in a team! In this case, you might have a server with a shared repository for your team, just like we have git-e.oru.se for our course!

The workflow in this case is fairly the same, when it comes to commits and branches. And you always work on a local copy of the shared repository. To get that local copy for the first time, you use *git clone* command. It pulls everything from the existing repository to your machine. Then, you can create branches, make commits, merge and all other things on your local repository. But to share what you have done with your team mates, you need to push your changes on server. So the command to do it is *git push*. Finally, your team mates need to use *git pull* to get new changes from the shared repository in the server.

Dev Process

Usually, you daily workflow starts with pulling recent updates from the server. You need to do it at least once a day when you come to work, so you can see what has been changed. Then, you make commits when you develop something new. The frequency of commits varies from team to team, but I usually do not recommend to have long (more than a day of work) commits, becasue they tend to become very heavy. Then, you push you code when you leave from work, or more often, depending on what you agree with your team.

In the real life software development, you always have more than one branch in your repository. Usually, master branch is kept for very stable production code and no one can commit to master directly. Master branch gets updated (merge requests) from some testing branch, which is manages by your testers to make sure that your code is doing what it is supposed to do. That branch can be called "testing" or "pre-production" os something else to reflect its purpose.

Finally, developers can keep one or more branches to develop individual features. When the

feature is finished, it is merged into testing branch.

Often, C++ repositories are configured to run tests and builds automatically (that is why some software has “nightly” builds). Therefore, if you commit unfinished or wrong code directly to testing or master branch, the build process will fail and you are guaranteed to have an exciting meeting with you team leader next morning.

Part 5: Practice

Equipped with a lot of knowledge on git workflow, you will now try to run it on a small repostory.

The task is fairly simple: you need to add a new source file to a dev branch in an existing repo with a function to perform some math calculations, create a test for your function, run it, commit everything to dev branch (if it actually works), then merge it to testing branch, test again with all other tests (if there are any) and if everything is good, merge to master.

So, clone the repository called “Git Workflow” to you machine and open project in NetBeans (Team – Remote – Clone...). Select all branches and choose dev branch to open after cloning. The project already contains something in main function and tests.

Add a new source file with a unique name (you can use your username). In the project view, right click on Source files and then Add new C++ source file. When adding a file, put it into math folder.

In the file, create a new function with the same unique name to do some math. But do not put the actuall math implementation yet!

Then, right click on your file and select Create Test – New C++ Simple Test, and select you function. Put some name to your test and again use some unique test name (e.g. your username-test). In the test function, fild and line saying something like

```
if (true /*check result*/) {...
```

This is the actual test condition, which is being tested. You need to add some values to the test and then check the result of your function execution is NOT equal to the correct answer. The idea is that the test should fail. In my case, I set `a=1`, `b=2` and then make a condition `result != 3`.

Save everything, make a clean build, right click on your test and select “Test”. It should fail. Write a body of your function to make the test pass, recompile, and test again.

When you get the test passing, right click on the project, Git – Commit. Write a meaningfull commit message and make sure that both, the source file and the test are included. Then Git-Remote-Push your local repo to the server.

After you pushed your development branch on the server, click on you project, Git – Branch/

Tag – Switch to Branch and select origit/testing.

Now, you need to merge the code which you commited to development into testing. While on testing branch, select Git – Branch/Tag - Merge Revision. Then click Select, choose development branch (or origin/development) and select the latest (the upmist) commit to merge. Now testing branch should contain all changes from development.

Run all test. Check if all tests pass. If everything looks good, merge testing branch into master in the same way as you've done before with development.

Done!